A Class of Heuristics for Reducing the Number of BWT-Runs in the String Ordering Problem

Gianmarco Bertola
Department of Computer Science, University of Pisa, Italy

Anthony J. Cox
Independent Researcher, Cambridge, UK

Veronica Guerrini[1]
Department of Computer Science, University of Pisa, Italy

Giovanna Rosone[2]
Department of Computer Science, University of Pisa, Italy

Abstract

The Burrows-Wheeler transform (BWT) is a famous text transformation that rearranges the symbols of the input strings so that occurrences of a same symbol tend to occur in runs. The number of runs is an important parameter in the BWT output string, historically associated with its high compressibility and more recently used as a measure for the space complexity of efficient data structures. It is a known fact that reordering the strings in the input collection $\mathcal{S}$ affects the number of runs in the output string $bwt(\mathcal{S})$ produced by applying the BWT to the string collection. In this paper, we define a class of transformed strings where symbols in particular blocks of the $bwt(\mathcal{S})$ can be reordered according to a different adaptive alphabet order. Then, we introduce new heuristics to reduce the number of runs in the BWT output of a string collection that improve on the two existing heuristics introduced in Cox et al. [7]. These new heuristics are computed when applying the BWT to a string collection assuming no *a priori* order on the input strings and without requiring any pre- and/or post- processing of the collection $\mathcal{S}$ or of the BWT string. In this paper, we also face the problem of reconstructing the input collection $\mathcal{S}$ from the string $\mathsf{bwt}(\mathcal{S})$ together with the string permutation realized when applying an alphabetical reordering of symbols during the construction of $\mathsf{bwt}(\mathcal{S})$.

2012 ACM Subject Classification Theory of computation → Data compression; Applied computing → Bioinformatics

Keywords and phrases Burrows-Wheeler Transform, SAP-interval, repetitive text, string compression

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.7

Funding Work partially supported by PNRR - M4C2 - Investimento 1.5, Ecosistema dell'Innovazione ECS00000017 - "THE - Tuscany Health Ecosystem" - Spoke 6 "Precision medicine & personalized healthcare", funded by the European Commission under the NextGeneration EU programme, by the MUR PRIN 2022YRB97K PINC and by INdAM - GNCS Project, codice CUP_E53C23001670001.

1 Introduction

The Burrows–Wheeler transform (BWT), introduced by M. Burrows and D. Wheeler in the 1990s [3] as a method for compressing a single input text, has since evolved into a versatile tool with many applications well beyond its original purpose [23]. Just as examples, the BWT has been used as the building block for compact text indexing [8, 16, 17, 10], and for bioinformatics applications, *e.g.*, for sequence alignment [20], phylogenetic analysis [12], genome assembly [24] as well as for sequencing data compression [13].

[1] corresponding author
[2] corresponding author

Informally, the BWT is a text transformation that rearranges the symbols of an input string S into a string $\mathsf{bwt}(S)$, which is obtained by concatenating the symbols that precede the cyclic rotations of S once the rotations have been sorted into lexicographic order. An equivalent and faster way to build $\mathsf{bwt}(S)$ [3] is to sort the suffixes of a related string obtained by appending an end-marker symbol (usually \$) that is lexicographically smaller than any of the symbols in S but does not appear in S itself. Both ways have two important properties: *reversibility* and *clustering effect*.

The *reversibility* permits to invert the transformed string by reconstructing S, and allows to search patterns in S very efficiently. While the *clustering effect* describes the inner property of the BWT to carry occurrences of a given symbol to runs of equal consecutive symbols.

The more symbols can be grouped into runs of the same symbol, the better is the performance of compression techniques such as, for instance, run-length encoding (RLE) where a string is coded as a concatenation of pairs formed by the symbol c and the number of times c is repeated. The total number of runs of a same symbol in the BWT-string is usually referred to as r. Recently, the parameter r is increasingly appearing not only for data compression, but also for measuring the space requirement of BWT-based text indexing data structures (see for instance [16, 17, 10]). Therefore, a text containing a few long runs is easier to compress or index than a text having the same characters but organized into a greater number of shorter runs. An interested reader can find theoretical studies and applications about the clustering effect in [19, 21, 22, 5] and references therein. Due to the ever-increasing volume and repetitive nature of data, developing new techniques that reduce or minimize the number of runs produced by the BWT is paramount for managing big data in applications.

Just as for a single string, the BWT of a collection of strings can be constructed either by sorting their cyclic rotations[3] as in [18] or sorting their suffixes [1]. In the latter case, a distinct end-marker symbol is appended to each string, making the collection ordered according to the order established among the end-marker symbols. Moreover, it is known that given a string collection $\mathcal{S}$, the two strings $\mathsf{bwt}(\mathcal{S})$ and $\mathsf{bwt}(\mathcal{S}')$ can only differ within particular intervals, if $\mathcal{S}'$ is a string permutation of $\mathcal{S}$ [7, 5].

Our contributions. In this paper, we define a class of transformed strings obtained by applying the BWT to a string collection $\mathcal{S}$ in which the symbols in particular blocks of the $\mathsf{bwt}(\mathcal{S})$ can permute according to a different adaptive alphabet ordering, while maintaining the reversibility property. Some known strategies falling into this class have already been introduced in the literature [7, 15, 2, 4]; and we recall them in Section 3.

Then, we introduce new heuristics for reducing the number of runs while computing the BWT-string; these heuristics improve on the number of runs of both the BWT-string obtained from the input-ordered collection and the two previously-introduced heuristics [7]. We show experimentally that the new heuristics tend to minimize the number of runs.

In this paper, the BWT output string is obtained by sorting all the suffixes of the input strings assuming that each string ends with a different end-marker symbol, but no *a priori* ordering of the end-marker symbols is given. Among the state-of-the-art approaches to compute the BWT for a string collection, we employ the algorithm BCR described in [1] to ensure that the order between any two end-marker symbols is determined during the construction of the BWT and not *a priori*. The interested reader can refer to [5] for a survey on the different output strings obtained by different tools.

[3] In this case, one needs to use the ω-order defined in [18].

We also address the problem of inverting the $\mathsf{bwt}(\mathcal{S})$ preserving the input order in $\mathcal{S}$ in case a symbol reordering has been applied during its construction. This property allows to reconstruct only a single string or groups of strings of the input collection and it might be useful in some applications, where only specific groups of strings are to be decoded (*e.g.*, in short-reads collections). In fact, without knowing the string reordering applied to $\mathsf{bwt}(\mathcal{S})$, the inverse transform of $\mathsf{bwt}(\mathcal{S})$ is no longer lossless in terms of string order.

1.1 Related works

In the literature, the problem of reducing the number of runs in the BWT-string has been approached from two perspectives. Indeed, on the one hand, the number of runs is affected by the order of the symbols in the considered alphabet; on the other hand, it is also impacted by the order of the strings in the collection.

Alphabet order. Chapin and Tate [6] show experimentally that ordering symbols by their ASCII code does not always give the best compression and discuss several heuristics for varying the alphabet order. For instance, they propose a scheme in which rotations are sorted in a manner inspired by reflected Gray codes. In [14], the authors introduce the Alternating BWT (ABWT) that is defined as the BWT by using a different order of the cyclic rotations, where one needs to alternate the standard and reverse orderings in odd and even positions. In [11], the authors describe a class of BWT string transformations based on context adaptive alphabet orderings, where in the rotation sorting phase, the alphabet orderings depend on the context (i.e., the longest common prefix of the rotations being compared). Moreover, they consider the problem of determining the BWT variant that minimizes the number of runs in the transformed string. Recently, Bentley et al. [2] derived the computational complexity of minimizing the number of runs in the BWT via alphabet ordering. They prove that the problem of deciding whether there exists an ordering of the alphabet symbols such that the number of runs in the BWT is at most equal to a given integer is NP-complete and its corresponding minimization problem is APX-hard.

String order. When the BWT is applied to a string collection by sorting the suffixes of its strings, one needs to append a different end-marker symbol to each string and to establish a order among them. In this case, the problem of minimizing the number of runs also needs to consider the different orderings of the input strings, since the ordering of the input strings depends on the reciprocal ordering of the end-marker symbols appended to each string. The authors in [7] provide the first experimental study showing: *i*) one can permute symbols within the $\mathsf{bwt}(\mathcal{S})$ associated to particular blocks, named “SAP-intervals” (SAP standing for *“same-as-previous”*), without destroying the string reversibility; *ii*) one can obtain a reduced number of runs in the $\mathsf{bwt}(\mathcal{S})$ while permuting symbols in SAP-intervals (see also [5]).

The problem of minimizing the number of runs in the $\mathsf{bwt}(\mathcal{S})$ via string ordering has been tackled as a closely related problem by Bentley et al. [2]. Indeed, finding a string order that minimizes the number of runs in the $\mathsf{bwt}(\mathcal{S})$ is equivalent to finding an order for the end-marker symbols that results in the minimum number of runs in the $\mathsf{bwt}(\mathcal{S})$. They show that given the $\mathsf{bwt}(\mathcal{S})$, the problem of minimizing its runs via string order can be solved in linear time by reducing such problem to a tuple sorting problem (more details in [2]). In [4], the authors provide the first implementation that computes the $\mathsf{bwt}(\mathcal{S})$ with the fewest number of runs using the post-processing strategy described in [2] combined with the SAP-array [7].

2 Preliminary and Materials

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet Σ with $c_1 < c_2 < \ldots < c_\sigma$, where $<$ denotes the standard lexicographic order. Let S be a string of length n on Σ. We denote the i-th symbol of S by $S[i]$. A *substring* of S is denoted as $S[i, j] = S[i] \cdot S[i + 1] \cdots S[j]$, with $\cdot$ being the concatenation operator.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of m strings on the alphabet Σ. We assume that each string $S_i \in \mathcal{S}$ has length $n_i + 1$, since we append a special end-marker symbol $\$_i$ to each S_i, *i.e.* $S_i[n_i + 1] = \$_i$, such that each $\$_i$ does not belong to Σ and it is lexicographically smaller than any other symbol in Σ. Let us denote by $N = \sum_{i=1}^{m} n_i + m$ the number of symbols of all strings in $\mathcal{S}$ (including their end-marker symbols).

The *suffix* of a string S_i starting at position k is $S_i[k, n_i + 1]$ and we define the j-suffix of S_i as the suffix starting at position $n_i + 1 - j$ of S_i, *i.e.* $S_i[n_i + 1 - j, n_i + 1]$, which has length $j + 1$ (including the end-marker symbol $\$_i$). Note that the *0-suffix* of S_i is just $\$_i$.

A *run* in a string S is a maximal substring consisting of repetitions of only one character.

The BWT is a reversible text transformation that, given as input a string $S\$$ (with $\$$ not appearing in S), produces an output string $\mathsf{bwt}(S\$)$ such that $\mathsf{bwt}[i]$ is the symbol preceding the i-th lexicographically smallest suffix of the string $S\$$. In the seminal paper by Burrows and Wheeler [3], two important properties that establish a correlation between the string $\mathsf{bwt}(S\$) = L$ and the string F, formed by lexicographically sorting the symbols of $S\$$, have been shown[4]:

- For all $i = 1 \ldots n + 1$, the symbol $F[i]$ circularly follows the symbol $L[i]$ in the string $S\$$;
- For each alphabet symbol c, the h-th occurrence of c in L corresponds to the h-th occurrence of c in F. In particular, if $L[i]$ is any occurrence of c in L, the position of its corresponding occurrence in F is given by $C[L[i]] + \mathrm{rank}(L[i], i)$, where $C[c]$ is the total number of symbols in $S\$$ that are smaller than c and $\mathrm{rank}(c, i)$ is the number of occurrences of c in the substring $L[1, i]$.

The above function that maps symbol occurrences in L to their corresponding symbol occurrences in F is known as *LF-mapping* [9].

2.1 The BWT applied to a string collection

A way for applying the BWT to a string collection consists in appending to each string an end-marker symbol and then concatenating the resulting strings to form a unique larger string. Nevertheless, it is also built without concatenating the input strings by using two approaches: *i)* sorting cyclic rotations of the input strings [18]; *ii)* sorting suffixes of the input strings [1]. The former approach uses a special order to sort the cyclic rotations which is not affected by the order of the input strings; while the sorting performed by the latter approach deeply depends on the order defined on the end-marker symbols. For this reason, in this paper, we focus on the latter approach and we follow the strategy introduced in [1] to handle the list of sorted suffixes. Note that in [1], the suffixes of the strings in $\mathcal{S}$ are sorted assuming that each string S_i ends with a distinct end-marker symbol $\$_i$ such that $\$_i < \$_j$, if $i < j$ in $\mathcal{S}$. See Table 1, sixth column (inputBWT), for an example of the BWT of a string collection $\mathcal{S}$ obtained by concatenating the symbols preceding the sorted suffixes of the strings in $\mathcal{S}$ (last column). The authors of [1] provide two related methods for computing such a BWT for large collections of strings making use of sequential reading and writing of

[4] The same properties hold for the BWT of a string collection [18, 1] - see Section 2.1.

files from disk: the first variant, BCR, is a semi external memory approach (see Section 3.1) that requires more RAM than the second variant, BEETL-BCRext, which uses negligible RAM at the expense of a larger amount of disk I/O.

2.2 SAP-array, SAP-interval and BWT by SAP

The authors of [7] showed that compression of the BWT output string can be improved by reordering the strings in the input collection, and that an "implicit sorting" strategy can be applied while computing the BWT. Such a strategy is based on the observation that in some particular blocks of the $\mathsf{bwt}(\mathcal{S})$, the order of the symbols is entirely determined by the order established among the associated j-suffixes that are equal up to the end-marker symbols [5]. In order to keep track of these blocks, we recall the notion of SAP-array and of SAP-interval.

▶ **Definition 1** ([7]). *The* SAP-array *(for 'same-as-previous'-array) of a collection $\mathcal{S}$ is a binary vector of the same length as the $\mathsf{bwt}(\mathcal{S})$ string such that $\mathsf{SAP}[i] = 1$ if and only if the suffix corresponding to the symbol $\mathsf{bwt}(\mathcal{S})[i]$ is* same as *the* previous *suffix in the list of sorted suffixes (their end-marker symbols excluded). A* SAP-interval *$\mathsf{bwt}(\mathcal{S})[b, e]$ is a maximal block of consecutive symbols such that $\mathsf{SAP}[i] = 1$, for all $b < i \leq e$.*

In other words, any run of 1's preceded by a 0 in the SAP-array corresponds to a block of equal j-suffixes, with $j \geq 0$, that differ only for their end-marker symbols.

Therefore, given two collections $\mathcal{S}$ and $\mathcal{S}'$ having the same strings but in different order, the following results hold (see [7, 5]).

▶ **Observation 2.** *The BWTs of $\mathcal{S}$ and $\mathcal{S}'$ have identical SAP-arrays and can only differ within SAP-intervals that contain more than one distinct symbol.*

▶ **Observation 3.** *Within a SAP-interval containing more than one distinct symbol, the reordering of the characters implicitly involves permuting the strings in the collection.*

3 A class of heuristics based on SAP-intervals

In this section, we describe a class of BWT transformed strings that reduce the number of runs based on the notion of SAP-intervals and the two key observations above (Observations 2 and 3). Moreover, we introduce new heuristics that apply an implicit string reordering during the construction of the BWT output string allowing a reduction of the number of runs with respect to the original input order.

We define the following class of transformed strings associated with a string collection $\mathcal{S}$:

▶ **Definition 4.** *Given a string collection $\mathcal{S}$, the class $\mathfrak{S}_{\mathcal{S}}$ comprises all the strings obtained from $\mathsf{bwt}(\mathcal{S})$ by possibly sorting the symbols of each SAP-interval according to a different adaptive alphabet ordering.*

The following existing variants of the BWT of $\mathcal{S}$ belong to the class $\mathfrak{S}_{\mathcal{S}}$:

1. **rloBWT** (or **colexBWT**), which is obtained by using the lexicographic alphabet ordering for each SAP-interval [15, 7] - see rloBWT column in Figure 1. Note that the rloBWT corresponds to sorting the input collection in reverse lexicographic order (RLO), or co-lexicographic order, and it can be computed not only by pre-processing the strings, but also on-the-fly during the construction of the $\mathsf{bwt}(\mathcal{S})$ itself (more details in [15]).

[5] Such a key observation is also stated in Bentley et al. [2], where such blocks are modeled as tuples, and in Cenzato et al. [5] through the notion of "interesting intervals".

2. **sapBWT**, which is obtained by sorting the symbols in those SAP-intervals whose number of distinct symbols is smaller than the SAP-interval's length, *i.e.* only in SAP-intervals in which it is possible to decrease the number of runs of the SAP-interval. The alphabet order used in any of such SAP-intervals, $\mathsf{bwt}[b, e]$, is given by setting $\mathsf{bwt}[b]$ as the smallest alphabet symbol and using the lexicographic order for all the other symbols. Note that the sapBWT is obtained on-the-fly during the construction of the $\mathsf{bwt}(\mathcal{S})$ (through BEETL-BCRext) where the SAP-array information is implicitly taken into account by computing a SAP status (more details in [7]) - see sapBWT column in Figure 1.
3. **optBWT**, which is obtained by using an alphabet order designed *ad hoc* for each SAP-interval containing more than one distinct symbol that minimizes the number of mismatches at the boundaries of the SAP-intervals. The *ad hoc* alphabet order for each SAP-interval is established in a backward fashion while scanning the $\mathsf{bwt}(\mathcal{S})$ and its SAP-array (both pre-computed) and using a stack to manage consecutive SAP-intervals (more details in [4]). In our running example (Figure 1), the optBWT is $TTTTTTT\$\$GGGG\$\$GGGGGCAAGC\$\$\$CCCAAAA$. Note that the number of runs in the optBWT is the minimum possible [2, 4].

Now, we focus on a particular subclass of $\mathfrak{S}_{\mathcal{S}}$ in which the adaptive alphabet order used in SAP-intervals is selected on-the-fly while building the BWT string itself. The sapBWT and rloBWT belong to this subclass, differently from the optBWT that is obtained as post-processing.

Therefore, we do not assume that the strings in $\mathcal{S}$ are ordered: we define a string order for $\mathcal{S}$ while building the BWT string, on the basis of the alphabet order choices performed in the SAP-intervals.

In Section 3.2, we define new heuristics belonging to $\mathfrak{S}_{\mathcal{S}}$ that reduce the number of runs on-the-fly during the construction of the BWT output string. To this end, we adopt the construction method introduced in [1] (see Section 3.1), which does not concatenate the input strings, but incrementally builds the $\mathsf{bwt}(\mathcal{S})$ by parsing the suffixes of the same length through a right-to-left scanning of all the strings at the same time.

3.1 BCR Construction and Data Structure Design

In this section we recall how the BCR algorithm works without describing the previous work in full detail, rather summarizing the explanation and data structures employed. For the space and time complexities of BCR we refer to the original article [1, Table 1].

BCR proceeds incrementally in k steps, where k is the length of the longest string in the collection plus one for the appended end-marker symbol. At the end of step j, BCR has built a partial BWT, $\mathsf{bwt}_j(\mathcal{S})$, corresponding to the concatenation of the symbols preceding the lexicographically sorted suffixes of length less than or equal to j.

In order to compute $\mathsf{bwt}_j(\mathcal{S})$, BCR needs an array A of m elements, that uses $O(m \log(m + |\Sigma| + |\mathsf{bwt}(\mathcal{S})|))$ bits of workspace, which is updated at each iteration j, for $j = 0, 1, 2, \ldots, k$. We denote by $A^{(j)}$ the array A at the j-th iteration, and by q any index in $[1, m]$, then:

- $A^{(j)}[q].seq$ stores the index of the string in $\mathcal{S}$ whose j-suffix is ranked q after lexicographically sorting all j-suffixes, *i.e.*, $A^{(j)}.seq$ gives the lexicographic order of all j-suffixes;
- $A^{(j)}[q].sym$ stores the symbol circularly preceding the j-suffix of the string with index $A^{(j)}[q].seq$, *i.e.*, a symbol to be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$;
- $A^{(j)}[q].pos$ stores in which position symbol $A^{(j)}[q].sym$ must be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$.

Table 1 The SAP-array and the different SAP-ordering heuristics applied to the collection $\mathcal{S} = \{CGAT, GGAT, CGCT, AGCT, AGAT, GGAT, GGCT\}$. The SAP-intervals are colored and the sorted suffixes related to $\mathcal{S}$ are listed in the last column. The number of runs is computed considering the end-marker symbols as the same symbol \$.

	Different heuristics string order						*Sorted suffixes in*
SAP-ARRAY	altBWT	plusBWT	randBWT	sapBWT	rloBWT	InputBWT	*input collection*
0	T	T	T	T	T	T	$\$_1$
1	T	T	T	T	T	T	$\$_2$
1	T	T	T	T	T	T	$\$_3$
1	T	T	T	T	T	T	$\$_4$
1	T	T	T	T	T	T	$\$_5$
1	T	T	T	T	T	T	$\$_6$
1	T	T	T	T	T	T	$\$_7$
0	\$	\$	\$	\$	\$	\$	$A\ G\ A\ T\ \$_5$
0	\$	\$	\$	\$	\$	\$	$A\ G\ C\ T\ \$_4$
0	G	G	G	G	G	G	$A\ T\ \$_1$
1	G	G	G	G	G	G	$A\ T\ \$_2$
1	G	G	G	G	G	G	$A\ T\ \$_5$
1	G	G	G	G	G	G	$A\ T\ \$_6$
0	\$	\$	\$	\$	\$	\$	$C\ G\ A\ T\ \$_1$
0	\$	\$	\$	\$	\$	\$	$C\ G\ C\ T\ \$_3$
0	G	G	G	G	G	G	$C\ T\ \$_3$
1	G	G	G	G	G	G	$C\ T\ \$_4$
1	G	G	G	G	G	G	$C\ T\ \$_7$
0	**A**	**G**	**C**	**C**	**A**	**C**	$G\ A\ T\ \$_1$
1	**C**	**G**	**A**	**A**	**C**	**G**	$G\ A\ T\ \$_2$
1	**G**	**A**	**G**	**G**	**G**	**A**	$G\ A\ T\ \$_5$
1	**G**	**C**	**G**	**G**	**G**	**G**	$G\ A\ T\ \$_6$
0	**G**	**C**	**G**	**C**	**A**	**C**	$G\ C\ T\ \$_3$
1	**C**	**G**	**A**	**A**	**C**	**A**	$G\ C\ T\ \$_4$
1	**A**	**A**	**C**	**G**	**G**	**G**	$G\ C\ T\ \$_7$
0	\$	\$	\$	\$	\$	\$	$G\ G\ A\ T\ \$_2$
1	\$	\$	\$	\$	\$	\$	$G\ G\ A\ T\ \$_6$
0	\$	\$	\$	\$	\$	\$	$G\ G\ C\ T\ \$_7$
0	A	A	C	A	A	A	$T\ \$_1$
1	A	A	C	A	A	A	$T\ \$_2$
1	A	A	C	A	A	C	$T\ \$_3$
1	A	A	A	A	A	C	$T\ \$_4$
1	C	C	A	C	C	A	$T\ \$_5$
1	C	C	A	C	C	A	$T\ \$_6$
1	C	C	A	C	C	C	$T\ \$_7$
Number of runs	13	12	13	14	14	17	

A trivial "iteration 0" sets the initial partial BWT, $\mathsf{bwt}_0(\mathcal{S})$, by simulating the insertion of the end-marker symbols in the sorted list of suffixes. Thus, we set $A^{(0)}[q].seq = q$, $A^{(0)}[q].sym = S_q[n_q]$ and $A^{(0)}[q].pos = q$, for $q = 1 \ldots m$. In the original version of BCR, $\mathsf{bwt}_0(\mathcal{S}) = A^{(0)}[1].sym \cdots A^{(0)}[m].sym$, *i.e.*, $\mathsf{bwt}_0(\mathcal{S})$ is the concatenation of the symbols preceding the end-marker symbols assuming that $\$_i < \$_j$, if $i < j$.

For each iteration $j = 1, 2, \ldots, k$, BCR computes $\mathsf{bwt}_j(\mathcal{S})$ by inserting the symbols preceding all the j-suffixes of $\mathcal{S}$ into $\mathsf{bwt}_{j-1}(\mathcal{S})$, through the following three phases:

1. BCR computes $A^{(j)}$ from $A^{(j-1)}$. For any q, let $x = A^{(j-1)}[q].seq$, $c = A^{(j-1)}[q].sym$ and $p = A^{(j-1)}[q].pos$. The value $A^{(j)}[q].pos$ is set by reading $\mathsf{bwt}_{j-1}(\mathcal{S})$ and by using the LF-mapping, *i.e.*, $A^{(j)}[q].pos = C[c] + rank(c, p)$ (we omit details for space reasons). While, $A^{(j)}[q].sym$ is updated with the symbol preceding the j-suffix of S_x.
2. BCR sorts the array $A^{(j)}$ by using $A^{(j)}.pos$ as sorting key.
3. For each q, BCR inserts the symbol $A^{(j)}[q].sym$ into $\mathsf{bwt}_{j-1}(\mathcal{S})$ at position $A^{(j)}[q].pos$.

At the end of iteration j, after inserting all the symbols preceding the j-suffixes into $\mathsf{bwt}_{j-1}(\mathcal{S})$, we get $\mathsf{bwt}_j(\mathcal{S})$ available for the next iteration. Whenever the first symbol of a string S_x has been inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$, the symbol $\$_x$ must be inserted at the next iteration and then no other symbol of the string S_x will be inserted.

After the last iteration k, all end-marker symbols have been inserted in their correct positions into $\mathsf{bwt}_{k-1}(\mathcal{S})$ and the BWT of the collection is completed.

Note that, the BCR implementation inserts the same end-marker symbol, \$, for all strings (*i.e.*, $\$_i = \$$ for all $i = 1, \ldots, m$) so as not to increase the size of the alphabet. However, one can store in a separate file the values in $A^{(j)}.seq$ to which each end-marker symbol is associated.

Actually, in BCR as well as in our implementation, the partial BWT is split into σ segments $B_j(z)$ formed by the symbols preceding suffixes starting with the symbol z (more details in [1]). Hence, in the array $A^{(j)}$, for each value $A^{(j)}[q].pos$, one needs to store two pieces of information: the symbol z and the position in $B_j(z)$ – see also [15].

3.2 Improved SAP-heuristics

Here we introduce three heuristics whose associated BWT strings belong to the class $\mathfrak{S}_{\mathcal{S}}$ of Definition 4. These heuristics are such that:

- they improve the number of runs of the BWT output string with respect to the input order (inputBWT) and the two existing heuristics sapBWT and rloBWT;
- symbols in SAP-intervals are sorted during an incremental construction of the BWT string that parses the suffixes of the same length through a simultaneous right-to-left scanning of all the strings, like BCR does. At the jth-iteration, the sorting takes into account symbols already stored in $\mathsf{bwt}_{j-1}(\mathcal{S})$ or that will be inserted into the $\mathsf{bwt}_j(\mathcal{S})$.

Let j be any BCR iteration, for $j = 1, \ldots, k$.

The first heuristic, called **altBWT**, uses an alternating lexicographic order to sort symbols in consecutive SAP-intervals which are to be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$. Thus, it differs from the rloBWT, as it alternates the lexicographic order and its inverse when inserting consecutive SAP-intervals - see altBWT column in Table 1.

The second heuristic, called **plusBWT**, designs an *ad hoc* alphabet order for each SAP-interval to be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$ on the basis of the symbols already in it. In particular, let p be the position in which the first symbol of the SAP-interval must be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$. We modify the alphabet order by setting the smallest symbol as $\mathsf{bwt}_{j-1}[p-1]$ (if it exists) and the greatest one as $\mathsf{bwt}_{j-1}[p]$ (if it exists), and by keeping the alphabet order among all the other symbols - see plusBWT column in Table 1.

The third heuristic, called **randBWT**, applies a random alphabet order for each SAP-interval inserted into bwt_{j-1}. Note that all these heuristics correspond to a string reordering that cannot be obtained *a-priori* unless having the associated string permutation.

In order to sort symbols according to any of the above strategies, the array $A^{(j)}$ defined in Section 3.1 is augmented with a binary value $A^{(j)}[q].sap$ that stores, for any $q = 1, \ldots, m$, the SAP-status of the associated symbol $A^{(j)}[q].sym$[6]. More precisely, $A^{(j)}[q].sap$ encodes whether or not the j-suffix of the string S_x, where $x = A^{(j)}[q].seq$, is same as the j-suffix of the string S_y, where $y = A^{(j)}[q-1].seq$, up to the end-marker symbol[7].

[6] Differently from [7], we compute the SAP-status and the SAP-intervals for the current iteration.
[7] Similar strategies have been used in [7, 15] and in the BCR-implementation of the tool `optimalBWT` introduced in [4] for explicitly computing the SAP-array.

Hence, if we have $A^{(j)}[q'].sap = 0$ and $A^{(j)}[q'+i].sap = 1$, for some q' and all i with $1 \leq i < \ell$, then there exist ℓ j-suffixes in $\mathcal{S}$ that are equal up to the end-marker symbols and that belong to the strings with indices $A^{(j)}[q'+i].seq$, for $0 \leq i < \ell$. For this reason, the symbols $A^{(j)}[q'+i].sym$, for $0 \leq i < \ell$, form a SAP-interval and they are inserted in consecutive positions into $\mathsf{bwt}_{j-1}(\mathcal{S})$.

Now, we describe how to modify BCR to compute any of the above heuristics.

At "iteration 0", the array A is initialized as described in Section 3.1. Moreover, since we simulate the insertion of the 0-suffixes of $\mathcal{S}$, we set $A^{(0)}[1].sap = 0$ and $A^{(0)}[q].sap = 1$, for all $1 < q \leq m$. Differently from the original BCR, before storing in $\mathsf{bwt}_0(\mathcal{S})$ the symbols $A^{(0)}[q].sym$, for $q = 1 \ldots m$, we perform a sorting on $A^{(0)}$ with respect to $A^{(0)}[\cdot].sym$ as sorting key. Supposing $\sigma < m$, we perform a linear sorting on $A^{(0)}$ on the basis of a special alphabet order, which is for both altBWT and plusBWT the alphabetic order, and for randBWT a random order on Σ. Then, by setting $\mathsf{bwt}_0(\mathcal{S}) = A^{(0)}[1].sym \ldots A^{(0)}[m].sym$, we have that the number of runs of $\mathsf{bwt}_0(\mathcal{S})$ is minimized. Note that the sorting is possible since we assume there is no fixed order among the end-marker symbols, (*i.e.*, it is no longer true that $\$_i < \$_j$ if $i < j$). The sorting of the symbols depends on the selected alphabet order rather than on the string ordering.

At each iteration $j = 1, 2, \ldots, k$, BCR updates the partial $\mathsf{bwt}_{j-1}(\mathcal{S})$ by inserting the symbols preceding the j-suffixes of $\mathcal{S}$ by using the three phases described in Section 3.1. We modify both phase 1 and phase 3 in order to update $A^{(j)}[q].sap$ values and to permute symbols in SAP-intervals while building $\mathsf{bwt}_j(\mathcal{S})$.

In particular, during phase 1, when computing $A^{(j)}$ from $A^{(j-1)}$, we propagate the SAP-status from iteration $j-1$ to iteration j. For each maximal interval $[b, e]$ in $A^{(j-1)}$ such that $A^{(j-1)}[i].sap = 1$ for all $b < i \leq e$, we set $A^{(j)}[q].sap = 0$ if $A^{(j-1)}[q].sym \neq A^{(j-1)}[q-1].sym$, for any $b < q \leq e$, and keep $A^{(j)}[q].sap = A^{(j-1)}[q].sap$, otherwise. Intuitively, let c and c' be the symbols preceding the two equal j-suffixes of S_x and S_y, where $x = A^{(j)}[q].seq$ and $y = A^{(j)}[q-1].seq$. Both c and c' are in the same SAP-interval, but being $c \neq c'$, the $(j+1)$-suffixes of S_x and S_y are no longer equal and thus their preceding symbols are no longer in the same SAP-interval. During phase 2, BCR sorts the array $A^{(j)}$ by using $A^{(j)}[q].pos$ as sorting key. No modifications need to be performed at this phase, but we can make a key observation relevant for phase 3: for each maximal interval $[b, e]$ in $A^{(j)}$ such that $A^{(j)}[i].sap = 1$ for $b < i \leq e$, the symbols $A^{(j)}[q].sym$ need to be inserted in consecutive positions into $\mathsf{bwt}_{j-1}(\mathcal{S})$ starting from position $p = A^{(j)}[b].pos$ (that is $A^{(j)}[b+i].pos = p+i$, for all $i = 1, \ldots, e-b$). During phase 3, for each maximal interval $[b, e]$ in $A^{(j)}$ such that $A^{(j)}[i].sap = 1$ for all $b < i \leq e$, we first linearly sort the sub-array $A^{(j)}[b, e]$ by using a specific alphabet order on the key $A^{(j)}.sym$, and then for all $b \leq q \leq e$, we write the symbol $A^{(j)}[q].sym$ into $\mathsf{bwt}_{j-1}(\mathcal{S})$ in consecutive positions starting from p.

At the end, BCR has built a BWT string for the collection $\mathcal{S}$ in which the string order is not given *a priori*, but it has implicitly established during the BWT construction itself according to the alphabet order used within SAP-intervals.

The additional space required with respect to the original BCR is given by both the space for storing the SAP status in m bits and the space used for linearly sorting the elements in the SAP-intervals, which is $O(\sigma \log m)$ bits to store the number of symbol occurrences in a SAP-interval and $O(m \log(\sigma + m))$ bits to linearly sort at most m symbols carrying the indices of the strings to which they belong. The time complexity of each iteration increases by $O(m)$, since first the array $A^{(j)}$ is scanned to find any maximal interval $[b, e]$ such that $A^{(j)}[i].sap = 1$ (for $b < i \leq e$) and for each of them the elements in $A^{(j)}[b, e]$ are linearly sorted according to $A^{(j)}.sym$. Thus, the overall space and time complexity remains as in [1, Table 1].

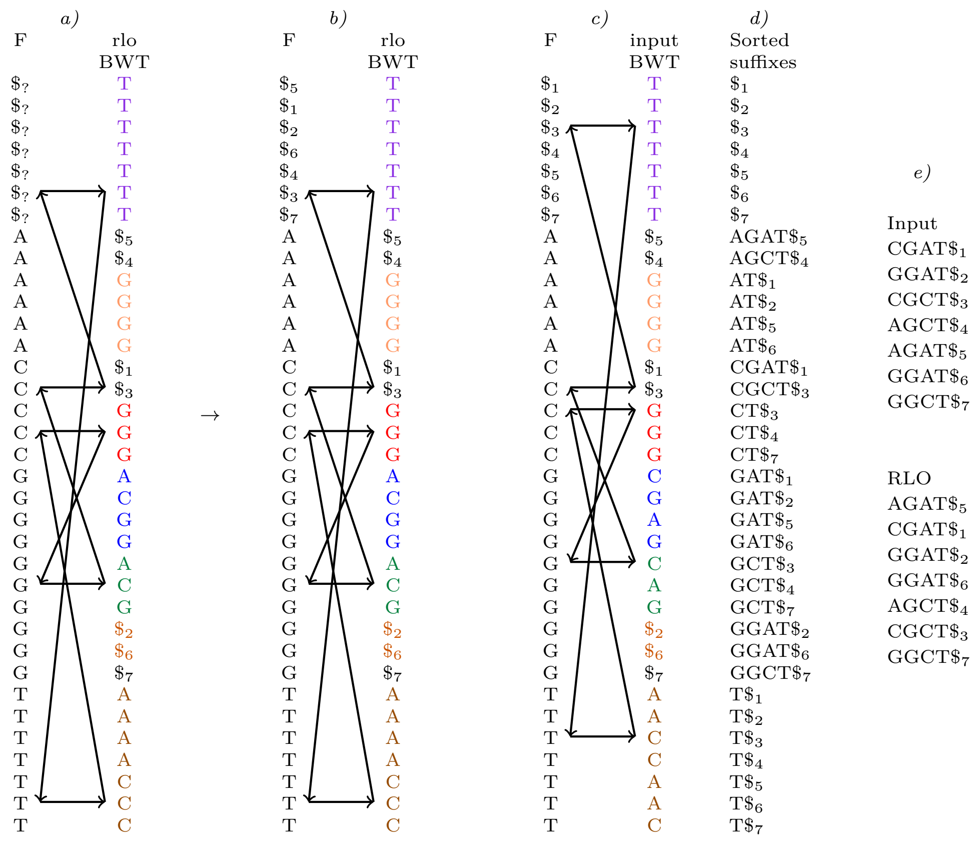
a)
F
rlo BWT
$_?$ $_?$ $_?$ $_?$ $_?$ $_?$ $_?$ A A A A A A C C C C C G G G G G G G G G G T T T T T T T
T T T T T T T $_5$ $_4$ G G G G $_1$ $_3$ G G G A C G G A C G $_2$ $_6$ $_7$ A A A A C C C
b)
F
rlo BWT
$_5$ $_1$ $_2$ $_6$ $_4$ $_3$ $_7$ A A A A A A C C C C C G G G G G G G G G G T T T T T T T
T T T T T T T $_5$ $_4$ G G G G $_1$ $_3$ G G G A C G G A C G $_2$ $_6$ $_7$ A A A A C C C
c)
F
input BWT
$_1$ $_2$ $_3$ $_4$ $_5$ $_6$ $_7$ A A A A A A C C C C C G G G G G G G G G G T T T T T T T
T T T T T T T $_5$ $_4$ G G G G $_1$ $_3$ G G G C G A G C A G $_2$ $_6$ $_7$ A A C C A A C
d)
Sorted suffixes
$_1$ $_2$ $_3$ $_4$ $_5$ $_6$ $_7$ AGAT$_5$ AGCT$_4$ AT$_1$ AT$_2$ AT$_5$ AT$_6$ CGAT$_1$ CGCT$_3$ CT$_3$ CT$_4$ CT$_7$ GAT$_1$ GAT$_2$ GAT$_5$ GAT$_6$ GCT$_3$ GCT$_4$ GCT$_7$ GGAT$_2$ GGAT$_6$ GGCT$_7$ T$_1$ T$_2$ T$_3$ T$_4$ T$_5$ T$_6$ T$_7$
e)
Input
CGAT$_1$ GGAT$_2$ CGCT$_3$ AGCT$_4$ AGAT$_5$ GGAT$_6$ GGCT$_7$
RLO
AGAT$_5$ CGAT$_1$ GGAT$_2$ GGAT$_6$ AGCT$_4$ CGCT$_3$ GGCT$_7$

Figure 1 Considering the string collection of Table 1: *a)* and *b)* show the decoding of a string when the strings are sorted by using the reverse lexicographic order (RLO). In a), the indices of the end-markers in column F cannot be assigned, if we do not know the string permutation performed during the encoding. In b), these indices are assigned since the encoding transformation outputted the string permutation. While *c)* shows the decoding of a string when no string permutation is performed. In *d)*, we list the sorted suffixes according to the input order, and in *e)* the two considerd re-orderings of our string collection.

4 Inverting the BWT and input order-preserving

In this section, we address the problem of inverting the BWT transform when a symbol re-ordering in the SAP-intervals has been applied.

For ease of description, Figures 1*a)-b)* show the columns F and L of the collection of our running example where the symbols of any SAP-interval have been sorted lexicographically (rloBWT). However, what we show in the following holds for all the other SAP-ordering heuristics. Figure 1*c)* shows the LF mapping applied to $S_3 = CGCT\$_3$ in the BWT string with input order of the collection, whose associated list of sorted suffixes is in Figure 1*d)*.

Note that by applying the LF-mapping to the rloBWT, starting from the first m symbols in L, we retrieve the m strings of the collection but permuted according to the order determined by the local alphabet order used within the SAP-intervals (RLO in Figure 1*e)*). However, the input string permutation can be recovered: in fact, when applying LF-mapping starting from the q-th symbol in L (e.g., in Figure 1*a)* $q = 6$), we end up with the end-marker $\$_k$ (e.g., in Figure 1*a)* $\$_3$) which means that the string S_k (S_3) has been placed at the q-th position (6-th position) in the permuted string collection. Since the indices of the end-marker symbols in L can be stored in a dedicated file, any \$ in L can be associated with the correct string to which it belongs even if all end-marker symbols appearing in L are equal to \$. In this way, since the LF-mapping starting from the q-th symbol in L ends in $\$_k$, we can label the q-th end-marker symbol $\$_?$ in F with the index k (e.g., in Figure 1*a)* $F[6] = \$_3$).

The example in Figure 1 also shows that, although the operation of symbol swapping is not visible in any SAP-interval with a run of a same symbol, the symbols are (implicitly) swapped with respect to the inputBWT (e.g., the red G-symbols in the third SAP-interval of Figure 1*c)*, since $\$_3 > \$_4$ in rloBWT).

We point out that by using LF-mapping we can decode the entire collection, but decoding one single string or a specific group of strings in $\mathcal{S}$ is not possible. Indeed, in Figure 1*a)*, we do not know how to decode the sixth string of the input collection, since starting from $L[6]$ we end up decoding the third string. In addition, it is not possible to start from $\$_6$ in L and to apply the LF-mapping, since we are not able to map $\$_6$ in L to the corresponding $\$_?$ in F. Therefore, the crucial property of decoding only specific groups of strings that the BCR algorithm guarantees is compromised.

We address this issue by designing a strategy so that BCR can output the permutation of the string indices at the end of the BWT construction phase. In this way, it is possible to assign the correct index to any end-marker symbol in F, and decoding groups of strings without decoding the entire collection (Figure 1*b)*).

Let π be an array of length m storing the permutation of the string indices of the input collection. For instance, in Figure 1*b)*, where rloBWT is computed, the permutation $\pi = [5\ 1\ 2\ 6\ 4\ 3\ 7]$. Whereas, at the last iteration, the array $A^{(4)}.seq$ contains the indices $[5\ 4\ 1\ 3\ 2\ 6\ 7]$, which correspond to the indices of the end-marker symbols in L. Indeed, at the last iteration, $A^{(k)}.seq$ contains the indices of the strings according to their lexicographic order, regardless of the SAP-ordering heuristics used for building the BWT string.

Therefore, during the BWT construction, we need to keep track of the symbol swapping performed. We modify the BCR data structure so that some entries of the array A point to indices of π. More precisely, for any iteration j, we have a pointer $A^{(j)}[q].pi$ to a position in π whenever a symbol swapping may affect the entries of $A^{(j)}$ from position q, *i.e.*, $A^{(j)}[q].sap = 0$ and $A^{(j)}[q+1].sap = 1$. That allows to report in π any string index swapping due to a symbol swapping within a SAP-interval. In fact, the array $A^{(j)}$ is designed to assign to each symbol $A^{(j)}[q].sym$ the string index to which it belongs (*i.e.*, $A^{(j)}[q].seq$).

After "iteration 0", we have $A^{(0)}[1].pi$ points to $\pi[1]$ and we initialize $\pi[q]$ with the value $A^{(0)}[q].seq$, for all $1 \leq q \leq m$. At each iteration j, we update $A^{(j)}[q].pi$ during phase 1 at the same time as $A^{(j)}[q].sap$. In particular, if $A^{(j)}[q].sap$ is set to 0, for some q, then $A^{(j)}[q].pi$ points to $\pi[x]$, where x is obtained by moving the position pointed by $A^{(j)}[q'].pi$ (with q' the rightmost index preceding q such that $A^{(j)}[q'].sap = 0$) by the offset $q - q'$. During phase 3, we need to update π when a symbol swapping is performed. Thus, if $[b, e]$ is a maximal interval in $A^{(j)}$ such that $A^{(j)}[q].sap = 1$ for all $b < q \leq e$, and $\pi[x]$ is the entry of π pointed by $A^{(j)}[b].pi$, then we copy in $\pi[x, x+b-e]$ the values $A^{(j)}[b, e].seq$ after the symbol swapping in that interval. At the end of the BWT construction, π corresponds to the list of the indices of the end-marker symbols in F.

5 Experimental Results

In this section, we assess the performance of the introduced heuristics that we have integrated into BCR tool[8] implemented in C++ working in semi-external memory. To evaluate the performance, we have designed a series of tests on real-life datasets (see Table 2).

[8] Source code: `https://github.com/giovannarosone/BCR_LCP_GSA`.

Table 2 Real-life datasets together with the BWT length, the maximum string length and the number of strings. The column optBWT reports the minimum number of runs for each dataset, the column ρ stores the ratio between the sum of the lengths of all SAP-intervals and the number of runs in them and the column τ stores the ratio between the number of runs in the SAP-intervals and the number of different symbols in them (higher values in bold).

	Dataset	Description	BWT length	Max len.	Number of sequences	optBWT	ρ	τ
1	SRR7494928–30	*Epstein Barr Virus*	984,191,064	101	9,648,932	40,700,607	3.32	**35.51**
2	ERR732065–70	*HIV-virus*	1,345,713,812	150	8,912,012	11,539,661	10.98	15.57
3	SRR12038540	*SARS-CoV-2 RBD*	1,690,229,250	50	33,141,750	14,864,523	7.08	**25.46**
4	ERR022075_1	*E. Coli str. K-12*	2,294,730,100	100	22,720,100	71,203,469	1.46	11.38
5	SRR059298	*Deformed wing virus*	2,455,299,082	72	33,634,234	48,376,632	8.09	17.62
6	SRR065389–90	*C. Elegans*	14,095,870,474	100	139,563,074	921,561,895	1.56	8.15
7	SRR2990914_1	*Sindibis virus*	15,957,722,119	36	431,289,787	105,250,120	3.16	**129.84**
8	ERR1019034	*H. Sapiens*	123,506,926,658	100	1,222,840,858	10,860,229,434	1.82	7.58
9	*pdb_seqres*	*proteins*	241,121,574	16,181	865,773	16,829,629	5.51	5.20

For each dataset, we computed two parameters: ρ and τ. The former parameter is given by the ratio between the sum of the lengths of all SAP-intervals in the BWT string and the total number of runs in the SAP-intervals of the inputBWT and it can be considered as a repetitiveness measure in SAP-intervals. The latter parameter is given by the ratio between the total number of runs in the SAP-intervals of the inputBWT and the sum of the number of distinct symbols in each SAP-interval. The higher τ, the more the heuristics can reduce the number of runs in the SAP-intervals, since the alphabet ordering applied to any SAP-interval reduces its number of runs to the number of distinct symbols.

All tests were done on a DELL PowerEdge R750 machine, 24-core machine with 2 Intel(R) Xeon(R) Gold 5318Y 24C/48T CPUs at 2.10 GHz, with 960 GB. The system is Ubuntu 22.04.2 LTS.

In Table 3, we report the number of runs for the new heuristics plusBWT, altBWT and randBWT, and show they improve on BWT-string with input order (inputBWT), and the two previously-introduced heuristics rloBWT and sapBWT[9].

Recall that the sapBWT heuristic is built using BEETL-BCRext [1], which uses negligible RAM at the expense of a larger amount of disk I/O. Therefore, its computation requires more time than the other heuristics. In fact, all other BWT-strings are obtained by the BCR-based tool that works in semi-external memory by sequential reading and writing files on disk and requires more RAM (to store the array A) than BEETL-BCRext.

The heuristics altBWT, plusBWT, randBWT and rloBWT have similar performances: on the largest dataset of about 123 Gb containing more than a billion sequences, they required a time construction of about 17 hours and an internal memory usage of about 20GB. On the contrary, the inputBWT required a time construction of about 15 hours and a similar internal memory usage (about 20GB). Note that the optBWT is computed as post-processing [4] by taking about 19 hours due to the fact that it needs to explicitly compute the SAP-array.

The experimental results show that plusBWT is the heuristic that gives the fewest runs, improving on the number of runs in inputBWT by up to 97% and giving at least a 50% reduction in runs for all eight of the DNA sequence datasets (strings from the alphabet $\{A, C, G, N, T\}$ of the same length). For the last dataset, containing proteins (on an alphabet of 26 symbols of variable length), we observe that the reduction in the number of runs

[9] Note that the implementation of sapBWT requires strings of the same length – see `https://github.com/BEETL/BEETL`.

Table 3 Number of runs in the BWT-string without symbol reordering (inputBWT) compared to the number of runs for any heuristic being in the class $\mathfrak{S}_{\mathcal{S}}$ for each dataset in Table 2.

		Different heuristics string order				
	inputBWT	rloBWT	sapBWT	plusBWT	altBWT	randBWT
1	254, 663, 327	41, 730, 649	65, 040, 263	**41, 372, 530**	41, 592, 394	41, 599, 327
2	48, 727, 709	11, 941, 093	17, 662, 811	**11,766,827**	11, 858, 536	11, 872, 578
3	209, 136, 502	17, 026, 009	17, 949, 348	**15,226,766**	16, 014, 506	16, 626, 930
4	259, 821, 570	75, 846, 202	92, 304, 201	**74,529,428**	75, 239, 739	75, 332, 300
5	249, 873, 376	50, 495, 777	75, 142, 244	**49,619,150**	50, 207, 432	50, 302, 961
6	2, 251, 887, 226	968, 098, 124	1, 066, 534, 827	**954,489,749**	960, 811, 214	963, 741, 035
7	3, 313, 966, 937	109, 772, 697	188, 817, 402	**108,466,351**	109, 365, 518	109, 599, 875
8	23, 084, 021, 291	11, 312, 737, 256	12, 151, 830, 264	**11,179,873,104**	11, 250, 843, 471	11, 273, 506, 405
9	17, 971, 532	16, 862, 960	–	**16,848,496**	16, 861, 264	16, 861, 897

is smaller compared to the one obtained for larger datasets, since the overall number of SAP-intervals is smaller. In addition, for this particular dataset, only $96,814$ of its $24,055,929$ SAP-intervals have at least two distinct symbols[10], and reordering symbols in SAP-intervals can have an impact on the number of runs only if SAP-intervals have at least two distinct symbols.

Finally, the additional overhead for the computation of any BWT-string in the class $\mathfrak{S}_{\mathcal{S}}$ is negligible compared to the number of runs reduction obtained with respect to the inputBWT.

6 Conclusions and further work

In this paper, we defined from a theoretical viewpoint a class $\mathfrak{S}_{\mathcal{S}}$ of transformed strings obtained by applying the BWT to a string collection $\mathcal{S}$ in which the symbols in particular blocks (SAP-intervals) permute according to a different adaptive alphabet ordering. We showed that the symbol swapping is important to reduce the number of runs in the BWT-string with respect to the one computed using the string input order, and it can be performed while maintaining the reversibility property of the BWT.

From a practical viewpoint, we introduced some heuristics belonging to $\mathfrak{S}_{\mathcal{S}}$ that reduce the number of runs, while computing the BWT-string itself. These heuristics improve on both the BWT-string obtained from the input-ordered collection and the two previously-introduced heuristics in [7].

In the experiments, the heuristics in the class $\mathfrak{S}_{\mathcal{S}}$ showed a considerable reduction in the number of runs. For instance, for all datasets (apart from *pdb_seqres* dataset), plusBWT obtained a reduction in the number of runs of about 50%-96% with respect to the inputBWT. Such reordering strategies can be very useful for data compression and for data structures whose properties have a favourable dependence on a small number of runs. Furthermore, the experiments showed that good results in terms of number of runs can be obtained using a random alphabet order for any SAP-interval (*i.e.*, randBWT). That heuristic performs better than the rloBWT heuristic that establishes the lexicographic alphabet order for each SAP-interval. This is an intriguing fact that shows that picking random symbols to place at the borders of a SAP-interval can be better than always choosing the lexicographic order to sort them. Experimentally, the best results are obtained when the alphabet order choice in

[10] These SAP-intervals with at least two distinct symbols are associated with the interesting intervals introduced in [5].

SAP-intervals keeps track of the symbols immediately preceding/succeeding, as done in the plusBWT heuristic. In addition, we observe that a pre-processing reordering of the input strings in $\mathcal{S}$ can only be applied if the string reordering is known *a priori*, such as for the reverse lexicographic order; nevertheless, this condition does not universally apply.

From Observations 2 and 3, we can conclude that the size of the introduced class $\mathfrak{S}_{\mathcal{S}}$ is at most $m!$. However, strings that are equal keep their original order in $\mathcal{S}$ and not all permutations may be possible. As future work, we intend to study further the permutations in the class $\mathfrak{S}_{\mathcal{S}}$ taking into account also the permutation study related to the rloBWT in [5].

Finally, an interesting direction for further studies involves to determine how the other data structures related to BWT are affected by the symbol swapping, considering that the LCP-array is not affected, as well as the SAP-array.

References

1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. Source code: `https://github.com/BEETL/BEETL`.

2 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In *ESA 2020*, volume 173 of *LIPIcs*, pages 15:1–15:13, 2020. `doi:10.4230/LIPIcs.ESA.2020.15`.

3 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

4 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Data Compression Conference, DCC 2023*, pages 71–80. IEEE, 2023. Source code: `https://github.com/davidecenzato/optimalBWT`. `doi:10.1109/DCC55655.2023.00015`.

5 Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. In *CPM 2022*, volume 223 of *LIPIcs*, pages 25:1–25:18, 2022. `doi:10.4230/LIPIcs.CPM.2022.25`.

6 Brenton Chapin and Stephen R. Tate. Higher compression from the Burrows-Wheeler transform by modified sorting. In *DCC*, page 532, Washington, DC, USA, 1998. IEEE Computer Society.

7 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinform.*, 28(11):1415–1419, 2012. Availability: Code is part of the BEETL library, available as a github repository at `https://github.com/BEETL/BEETL`. `doi:10.1093/bioinformatics/bts173`.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000. `doi:10.1109/SFCS.2000.892127`.

9 Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001. `doi:10.1016/S0020-0255(01)00098-6`.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1), 2020. `doi:10.1145/3375890`.

11 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. A new class of string transformations for compressed text indexing. *Information and Computation*, 294:105068, 2023. `doi:10.1016/j.ic.2023.105068`.

12 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT2: phylogeny reconstruction via eBWT positional clustering. *Algorithms Mol. Biol.*, 18(1):11, 2023. `doi:10.1186/S13015-023-00232-4`.

13 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Parallel lossy compression for large FASTQ files. In *Biomedical Engineering Systems and Technologies*, pages 97–120, Cham, 2023. Springer Nature Switzerland. `doi:10.1007/978-3-031-38854-5_6`.

14 Christophe Reutenauer Ira M. Gessel, Antonio Restivo. A bijection between words and multisets of necklaces. *Eur. J. Combin.*, 33(7):1537–1546, 2012.

15 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014. Source code: `https://github.com/lh3/ropebwt2`. `doi:10.1093/bioinformatics/btu541`.

16 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.

17 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

18 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. `doi:10.1016/j.tcs.2007.07.014`.

19 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, and Luca Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017.

20 Joong Chae Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. Fm-index of alignment with gaps. *Theor. Comput. Sci.*, 710:148–157, 2018.

21 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021.

22 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2021.

23 Giovanna Rosone and Marinella Sciortino. The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words. In *CiE*, volume 7921 LNCS of *LNCS*, pages 353–364. Springer, 2013. `doi:10.1007/978-3-642-39053-1_42`.

24 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinform.*, 26(12):367–373, 2010.